Comunicación y sincronización entre tareas

Miguel Alfonso Castro García
mcas@xanum.uam.mx

Universidad Autónoma Metropolitana - Izt

3 de junio de 2020

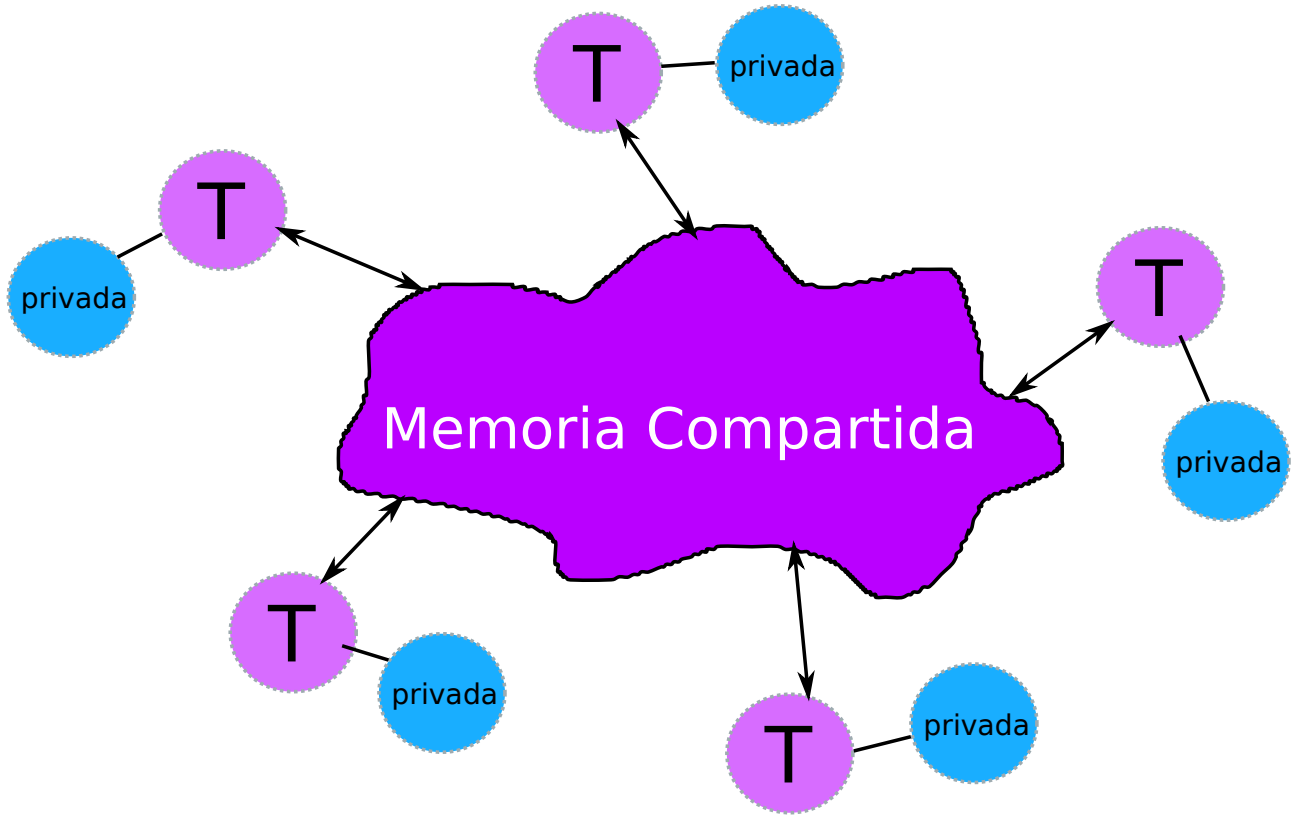
Contenido

Comunicación

Las tareas se comunican básicamente por dos formas:

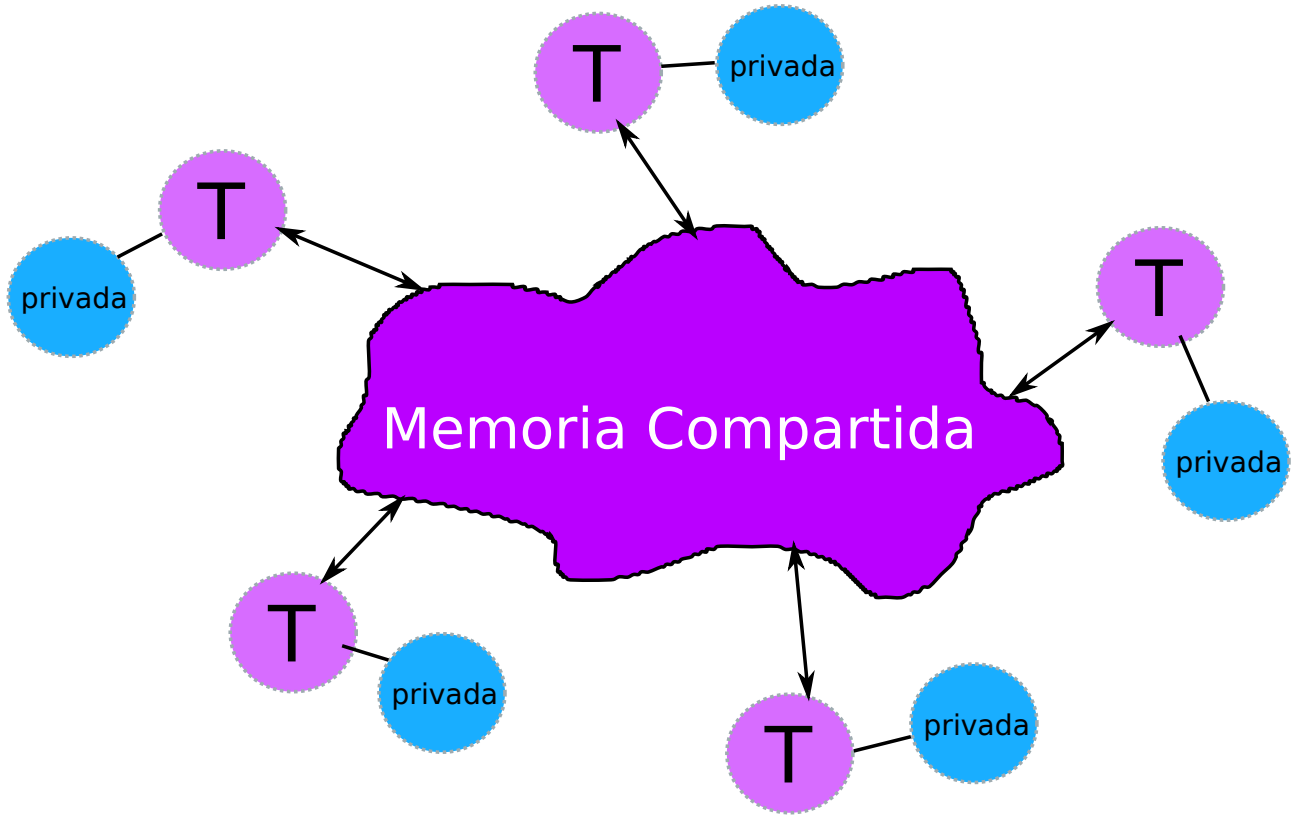
- Memoria compartida
- Paso de mensajes

Memoria compartida



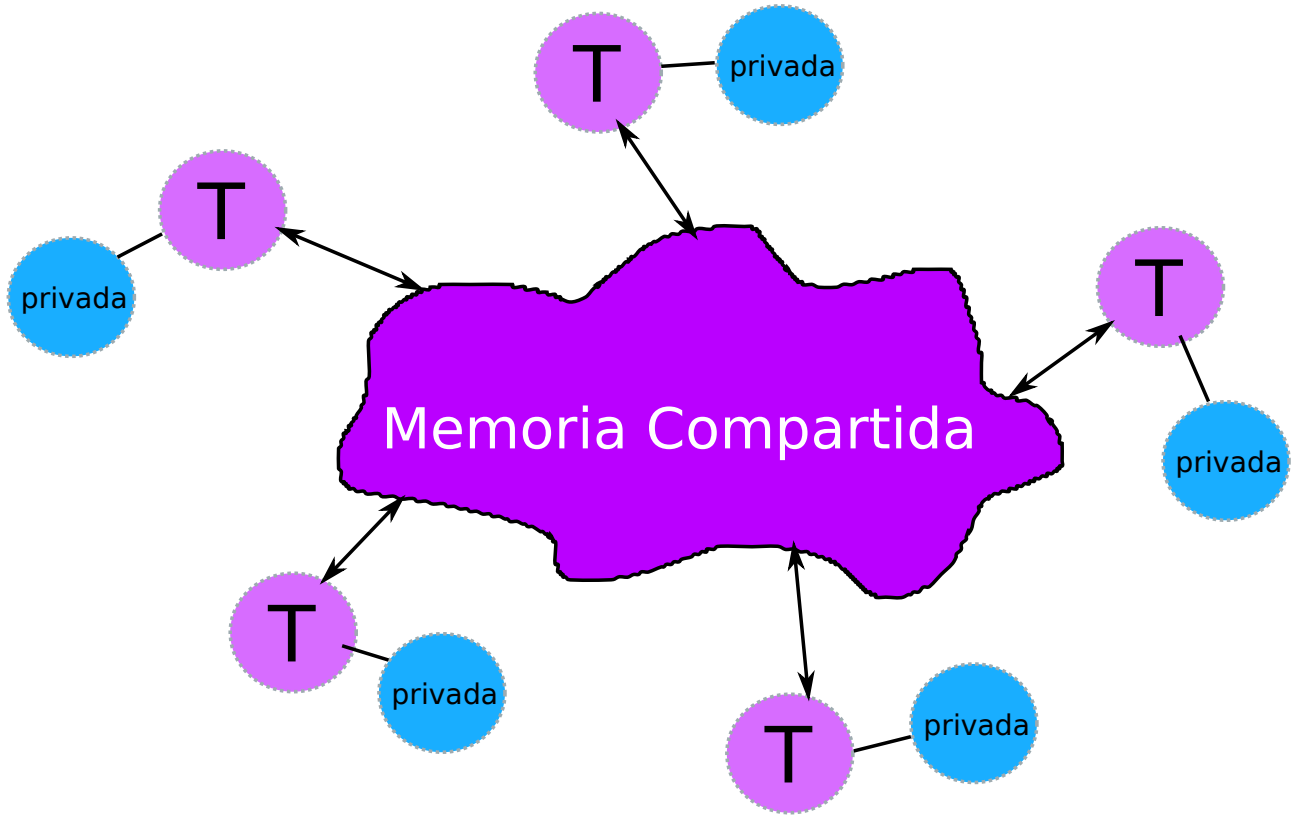
- La comunicación es **indirecta**
- Las tareas se comunican mediante la *lectura* y *escritura* en la memoria compartida

Memoria compartida



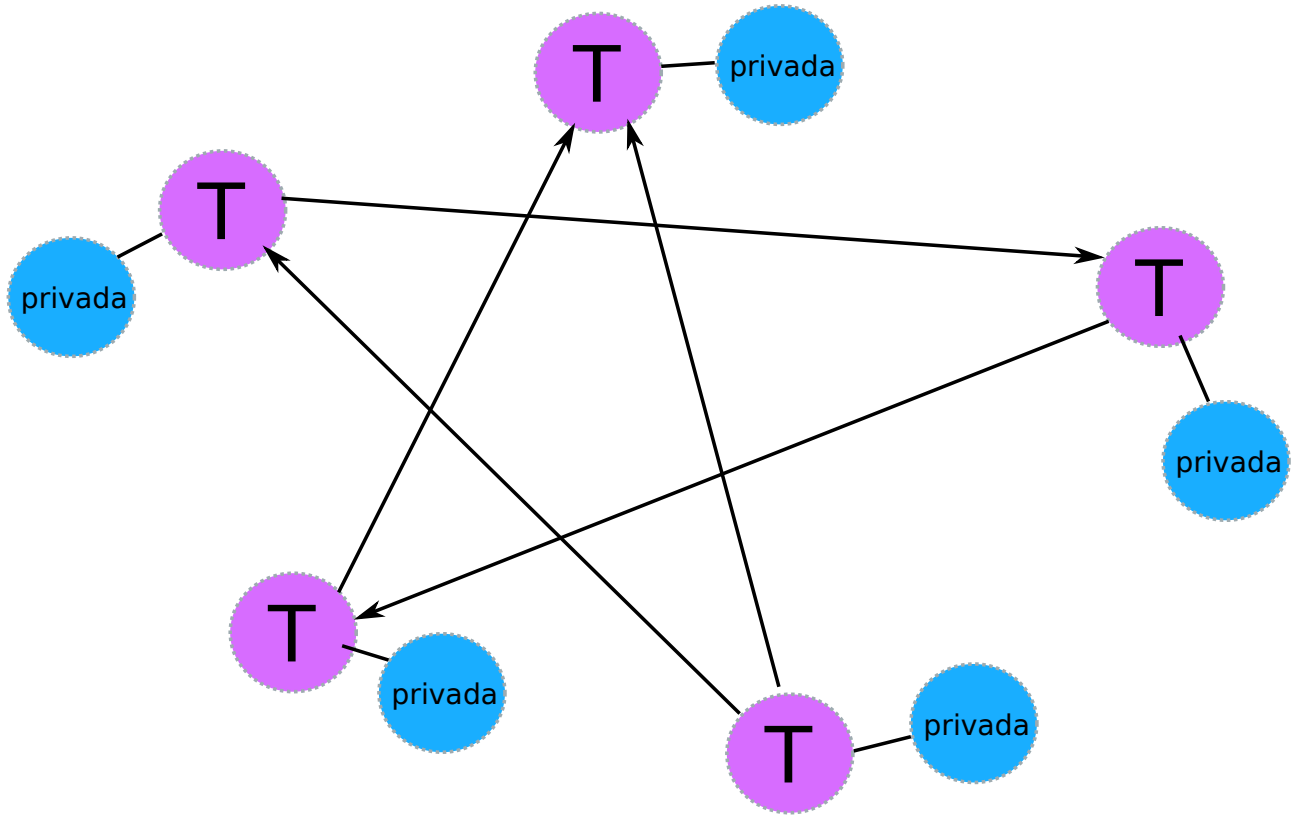
- La comunicación es **indirecta**
- Las tareas se comunican mediante la *lectura* y *escritura* en la memoria compartida

Memoria compartida



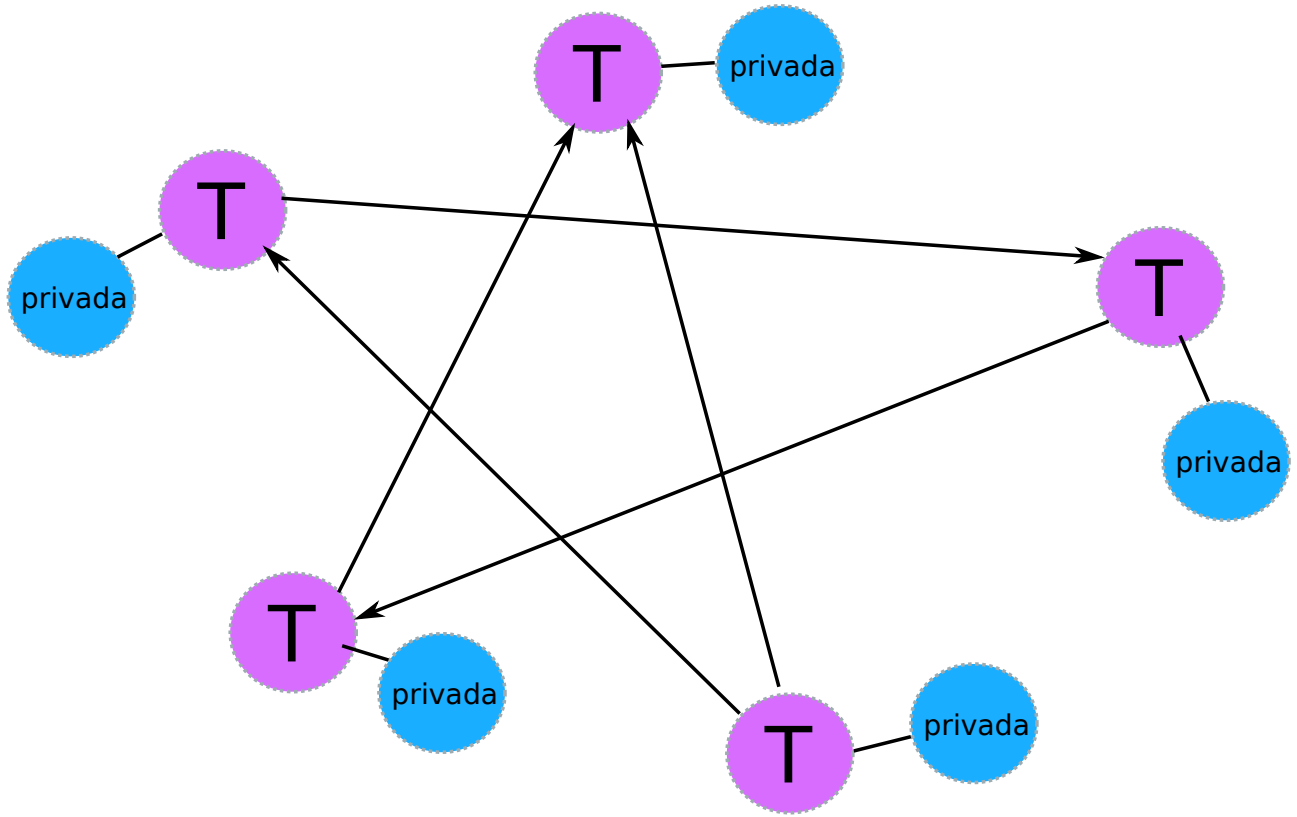
- La comunicación es **indirecta**
- Las tareas se comunican mediante la *lectura* y *escritura* en la memoria compartida

Paso de mensajes



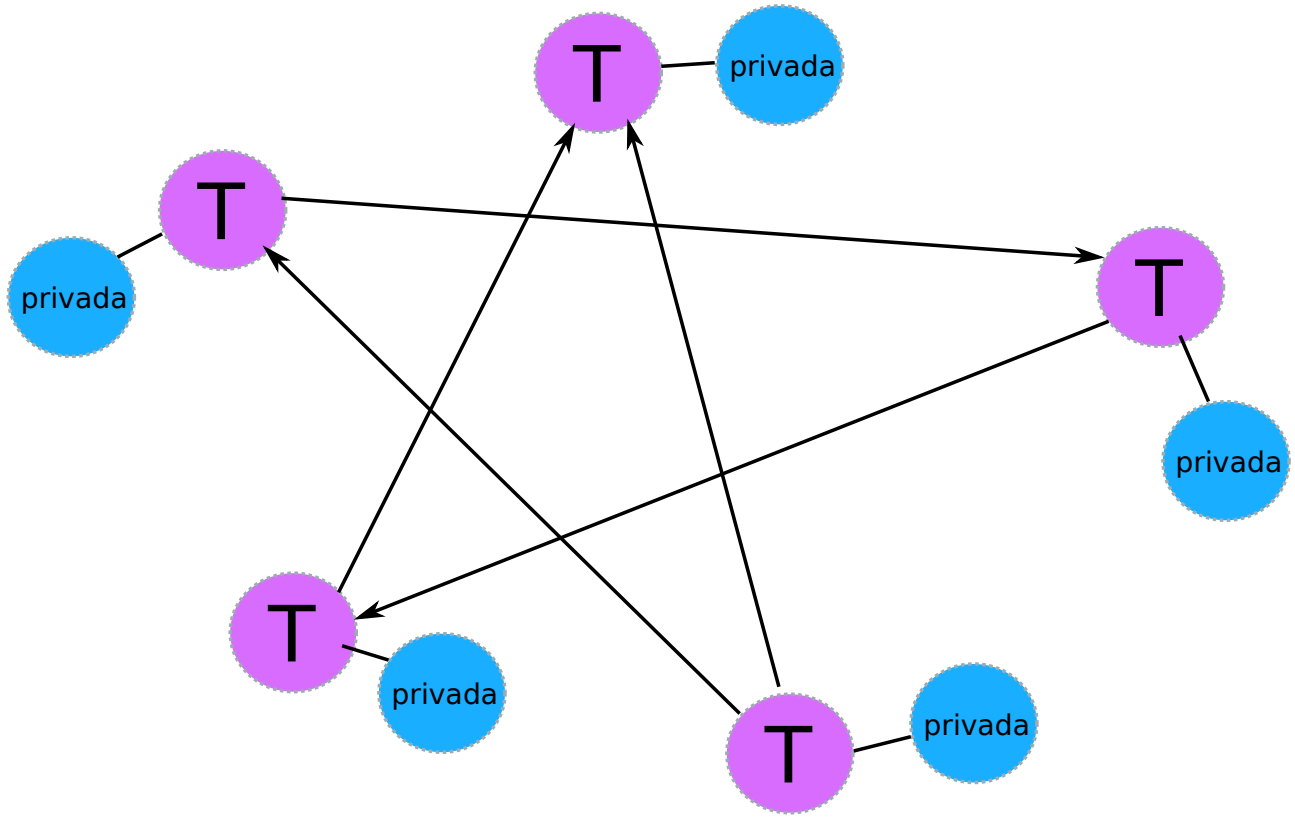
- La comunicación es **directa**
- Las tareas se comunican a través del *envío* y *recepción* de mensajes

Paso de mensajes



- La comunicación es **directa**
- Las tareas se comunican a través del *envío* y *recepción* de mensajes

Paso de mensajes



- La comunicación es **directa**
- Las tareas se comunican a través del *envío* y *recepción* de mensajes

Ejemplos

- Señales de humo
- Botella en el mar
- Teléfono
- Cartas
- SMS
- Twitter
- Alerta sísmica
- WWW

Ejemplos

- Señales de humo
- Botella en el mar
- Teléfono
- Cartas
- SMS
- Twitter
- Alerta sísmica
- WWW

Ejemplos

- Señales de humo
- Botella en el mar
- Teléfono
- Cartas
- SMS
- Twitter
- Alerta sísmica
- WWW

Ejemplos

- Señales de humo
- Botella en el mar
- Teléfono
- Cartas
- SMS
- Twitter
- Alerta sísmica
- WWW

Ejemplos

- Señales de humo
- Botella en el mar
- Teléfono
- Cartas
- SMS
- Twitter
- Alerta sísmica
- WWW

Ejemplos

- Señales de humo
- Botella en el mar
- Teléfono
- Cartas
- SMS
- Twitter
- Alerta sísmica
- WWW

Ejemplos

- Señales de humo
- Botella en el mar
- Teléfono
- Cartas
- SMS
- Twitter
- Alerta sísmica
- WWW

Ejemplos

- Señales de humo
- Botella en el mar
- Teléfono
- Cartas
- SMS
- Twitter
- Alerta sísmica
- WWW

Recordando la sincronización es necesaria para:

- Exclusión mutua
- Condición de sincronización

Algunos de los mecanismos que se dispone para hacer sincronización son:

1. Inhibición de interrupciones
2. Espera ocupada (busy waiting)
   - variables candado
   - alternancia estricta
   - tsl (test and set lock)
3. Mutex
4. Semáforos
5. Barreras
6. Operaciones de paso de mensajes send/receive

Inhibición de interrupciones

Solución de hardware

- Deshabilitar interrupciones justo antes de entrar a la región crítica y rehabilitarlas justo después de dejarla
- Deshabilitadas las interrupciones no hay interrupciones de reloj que ocurran

Inhibición de interrupciones

Solución de hardware

- Deshabilitar interrupciones justo antes de entrar a la región crítica y rehabilitarlas justo después de dejarla
- Deshabilitadas las interrupciones no hay interrupciones de reloj que ocurran

Inhibición de interrupciones

Solución de hardware

- Deshabilitar interrupciones justo antes de entrar a la región crítica y rehabilitarlas justo después de dejarla
- Deshabilitadas las interrupciones no hay interrupciones de reloj que ocurran

Inhibición de interrupciones

Solución de hardware

- Deshabilitar interrupciones justo antes de entrar a la región crítica y rehabilitarlas justo después de dejarla
- Deshabilitadas las interrupciones no hay interrupciones de reloj que ocurran

Desventajas

- dar el poder o control a los usuarios para deshabilitarlas
- si es multiprocesador el inhibir la interrupción sólo afecta al cpu que ejecuta la instrucción *disable*

Variables candado

Solución de software

- se considera una variable compartida (candado) inicialmente a cero
- cuando un proceso quiere entrar a la region crítica prueba el candado
  - si el candado es 0, lo coloca a 1 y entra
  - si el candado es 1 espera hasta que se vuelve 0

Variables candado

Solución de software

- se considera una variable compartida (candado) inicialmente a cero
- cuando un proceso quiere entrar a la region crítica prueba el candado
  - si el candado es 0, lo coloca a 1 y entra
  - si el candado es 1 espera hasta que se vuelve 0

Variables candado

Solución de software

- se considera una variable compartida (candado) inicialmente a cero
- cuando un proceso quiere entrar a la region crítica prueba el candado
  - si el candado es 0, lo coloca a 1 y entra
  - si el candado es 1 espera hasta que se vuelve 0

Variables candado

Solución de software

- se considera una variable compartida (candado) inicialmente a cero
- cuando un proceso quiere entrar a la region crítica prueba el candado
  - si el candado es 0, lo coloca a 1 y entra
  - si el candado es 1 espera hasta que se vuelve 0

Desventajas

- falla (mas de un proceso puede entrar a la región crítica)
- espera activa

Alternancia estricta

Una variable *turn* (inicialmente en 0) mantiene evidencia de quién es el turno para entrar a la región crítica y examinar o actualizar la región crítica

```
P0
while(TRUE) {
  while(turn!=0)  /* loop */
  region_critica();
  turno = 1;
  region_no_critica();
}
```

```
P1
while(TRUE) {
  while(turn!=1)  /* loop */
  region_critica();
  turno = 0;
  region_no_critica();
}
```

Alternancia estricta

Una variable *turn* (inicialmente en 0) mantiene evidencia de quién es el turno para entrar a la región crítica y examinar o actualizar la región crítica

```
P0
while(TRUE) {
  while(turn!=0)  /* loop */
  region_critica();
  turno = 1;
  region_no_critica();
}
```

```
P1
while(TRUE) {
  while(turn!=1)  /* loop */
  region_critica();
  turno = 0;
  region_no_critica();
}
```

Alternancia estricta

Una variable *turn* (inicialmente en 0) mantiene evidencia de quién es el turno para entrar a la región crítica y examinar o actualizar la región crítica

```
P0
while(TRUE) {
  while(turn!=0)  /* loop */
  region_critica();
  turno = 1;
  region_no_critica();
}
```

```
P1
while(TRUE) {
  while(turn!=1)  /* loop */
  region_critica();
  turno = 0;
  region_no_critica();
}
```

Desventajas

- si un proceso es más lento que otro falla

TSL (Test and Set Lock)

TSL RX, LOCK

- Copia el contenido de la palabra LOCK en RX y luego almacena una valor no-zero en LOCK
- La operación de lectura y almacenamiento se garantizan indivisibles

```
enter_region:
  TSL REGISTER,LOCK
  CMP REGISTER, #0
  JNE enter_region
  RET
```

```
leave_region:
  MOVE LOCK,#0
  RET
```

TSL (Test and Set Lock)

TSL RX, LOCK

- Copia el contenido de la palabra LOCK en RX y luego almacena una valor no-zero en LOCK
- La operación de lectura y almacenamiento se garantizan indivisibles

```
enter_region:
  TSL REGISTER,LOCK
  CMP REGISTER, #0
  JNE enter_region
  RET
```

```
leave_region:
  MOVE LOCK,#0
  RET
```

TSL (Test and Set Lock)

TSL RX, LOCK

- Copia el contenido de la palabra LOCK en RX y luego almacena una valor no-zero en LOCK
- La operación de lectura y almacenamiento se garantizan indivisibles

```
enter_region:
  TSL REGISTER,LOCK
  CMP REGISTER, #0
  JNE enter_region
  RET
```

```
leave_region:
  MOVE LOCK,#0
  RET
```

TSL (Test and Set Lock)

TSL RX, LOCK

- Copia el contenido de la palabra LOCK en RX y luego almacena una valor no-zero en LOCK
- La operación de lectura y almacenamiento se garantizan indivisibles

```
enter_region:
  TSL REGISTER,LOCK
  CMP REGISTER, #0
  JNE enter_region
  RET
```

```
leave_region:
  MOVE LOCK,#0
  RET
```

¿ Desventaja ?

Mutex

- Mutex es una abreviación de "exclusión mutua"
- Las variables *mutex* permiten implementar la sincronización entre threads, protegiendo los datos compartidos
- Una variable *mutex* actúa como un **candado** protegiendo el acceso a los datos o recursos compartidos

Las operaciones básicas:

Mutex

- Mutex es una abreviación de “exclusión mutua”
- Las variables *mutex* permiten implementar la sincronización entre threads, protegiendo los datos compartidos
- Una variable *mutex* actúa como un **candado** protegiendo el acceso a los datos o recursos compartidos

Las operaciones básicas:

Mutex

- Mutex es una abreviación de "exclusión mutua"
- Las variables *mutex* permiten implementar la sincronización entre threads, protegiendo los datos compartidos
- Una variable *mutex* actúa como un **candado** protegiendo el acceso a los datos o recursos compartidos

Las operaciones básicas:

Mutex

- Mutex es una abreviación de “exclusión mutua”
- Las variables *mutex* permiten implementar la sincronización entre threads, protegiendo los datos compartidos
- Una variable *mutex* actúa como un **candado** protegiendo el acceso a los datos o recursos compartidos

Las operaciones básicas:

Mutex

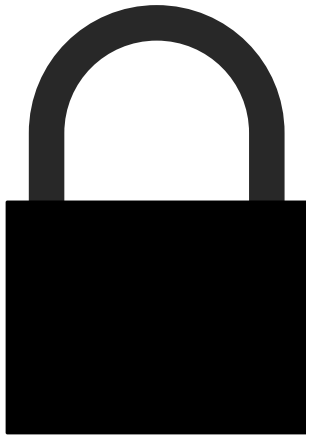
- Mutex es una abreviación de "exclusión mutua"
- Las variables *mutex* permiten implementar la sincronización entre threads, protegiendo los datos compartidos
- Una variable *mutex* actúa como un **candado** protegiendo el acceso a los datos o recursos compartidos

Las operaciones básicas:

Mutex

- Mutex es una abreviación de "exclusión mutua"
- Las variables *mutex* permiten implementar la sincronización entre threads, protegiendo los datos compartidos
- Una variable *mutex* actúa como un **candado** protegiendo el acceso a los datos o recursos compartidos

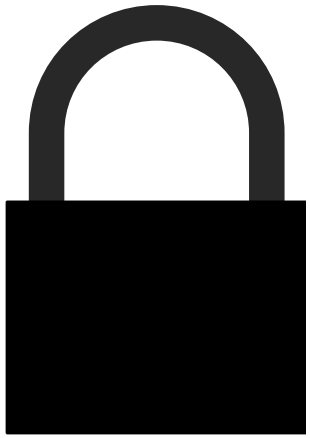
Las operaciones básicas:



Mutex

- Mutex es una abreviación de "exclusión mutua"
- Las variables *mutex* permiten implementar la sincronización entre threads, protegiendo los datos compartidos
- Una variable *mutex* actúa como un **candado** protegiendo el acceso a los datos o recursos compartidos

Las operaciones básicas:



Mutex

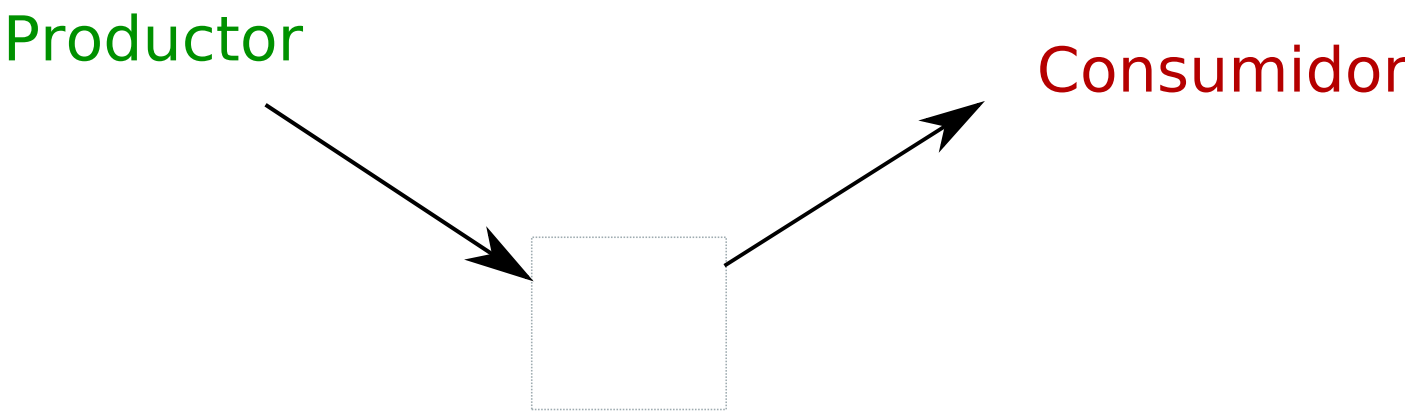
- El concepto básico de un mutex es que **sólo un thread** puede cerrar (o poseer) una variable mutex en un tiempo dado
- Si varios threads intentan cerrar un mutex, sólo uno de ellos tendrá exito. Ningún otro thread puede obtener el mutex hasta que el thread que lo tiene libere el mutex
- Los threads deben tomar turnos para acceder a los datos compartidos

Productor-Consumidor

Considera dos entes:

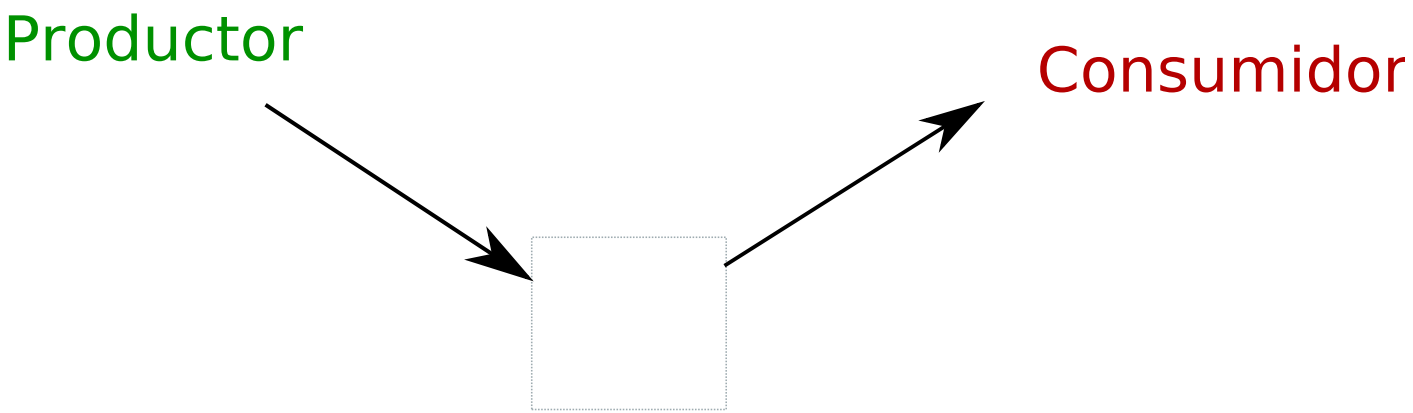
- El productor guarda o inserta elementos
- El consumidor toma o borra elementos

1 Productor - 1 Consumidor - 1 Casilla



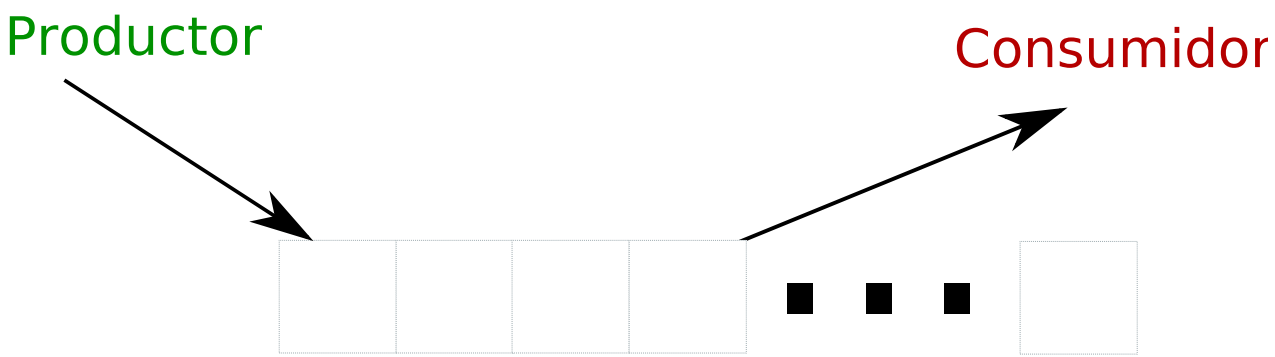
¿ Que pasa si hay mas casillas ?

1 Productor - 1 Consumidor - 1 Casilla



¿ Que pasa si hay mas casillas ?

1 Productor - 1 Consumidor - N Casillas



Solución con **sleep** and **wakeup**

```
#define N 100
int count = 0

void producer(void)
{
  int item;
  while(TRUE) {
  item = produce_item();
  if (count==N) sleep();
  insert_item(item)
  count=count+1;
  if(count==1)
      wakeup (consumer)
  }
}
```

```
void consumer(void)
{
  int item;
  while(TRUE) {
  if (count==0) sleep();
  item=remove_item();
  count=count-1;
  if(count==N-1)
      wakeup (producer)
  consume_item();
  }
}
```

Semáforos

En 1965 Dijkstra propuso el concepto de abstracción de un semáforo para el manejo de exclusión mutua y sincronización

Semáforo

Es un número entero con dos operaciones atómicas: *wait* y *signal/post*, también llamadas *down* y *up*

Operación down

- La operación *down* checa si el valor es mas grande que cero, si es así decrementa el valor
- Si el valor el cero el proceso se pone a **dormir** sin completar el down por el momento
- Checar el valor, cambiarlo y posiblemente el irse a dormir es hecho en un solo paso, indivisible (atómico)
- Una vez que la operación ha comenzado ningún otro proceso puede acceder al semáforo (hasta que la operación se ha completado o bloqueado)

Operación up

- La operacion *up* incrementa el valor del semáforo direcccionado
- Si uno o más procesos estuvieron durmiendo en dicho semáforo (incapaces de completar una operación down previa) uno de ellos es elegido por el sistema (aleatorio) y le es permitido completar su operación down
- La operación de incrementar un semáforo y posiblemente despertar algún proceso es también **indivisible**

Problema del productor-consumidor usando semáforos

La solución usa 3 semáforos:

1. **full** - para contar el número de slots (casillas) que están llenas
2. **empty** - para contar el número de slots que están vacías
3. **mutex** - para estar seguro que el productor y consumidor no acceden al buffer al mismo tiempo

Problema del productor-consumidor usando semáforos

```
#define N 100
typedef int semaphore
semaphore mutex = 1;
semaphore empthy = N;
semaphore full = 0;

void producer(void) {
  int item;
  while(TRUE) {
  item = produce_item();
  down(&empthy)
  down(&mutex)
  insert_item(item);
  up(&mutex);
  up(&full);
```

```
void consumer(void) {
  int item;
  while(TRUE) {
  down(&full)
  down(&mutex)
  item=remove_item();
  up(&mutex);
  up(&empthy);
  consume_item();
  }
}
```